Trusted Components
Reuse, Contracts and Patterns

Prof. Dr. Bertrand Meyer
Dr. Karine Arnout
Lecture 18: Builder, Proxy, State
Agenda for today

- Builder
- Proxy
- State
  - Original pattern and variants
  - Componentization
"Separate the construction of a complex object from its representation so that the same construction process can create different representations." [GoF, p 97]
Componentizability classification

1. Componentizable
   1.1 Built-in
   1.2 Library-supported
   1.3 Newly componentized
   1.4 Possible component

   1.3.1 Fully componentizable
       Prototype

   1.3.2 Componentizable but not comprehensive
       Flyweight
       Observer
       Mediator
       Abstract Factory
       Factory Method
       Visitor
       Command
       Composite
       Chain of Responsibility

   1.3.3 Componentizable but unfaithful
       Strategy

   1.3.4 Componentizable but useless
       Memento

2. Non-componentizable
   2.1 Skeleton
       Singleton
       Facade
       Interpreter

   2.2 Possible skeleton
       Iterator
       Bridge

   2.3 Some library support
       Adapter

   2.4 Design idea
       Template Method
       Decorator

Design pattern

Componentizable
Non-componentizable

Fully componentizable
Componentizable
Componentizable but not comprehensive
Componentizable but unfaithful
Componentizable but useless
No method
Method

deferred class BUILDER [G]

feature -- Access
last_product: G is
  -- Product under construction
deferred
end

feature -- Status report
is_ready: BOOLEAN is
  -- Is builder ready to build last_product?
deferred
end

feature -- Basic operations
build is
  -- Build last_product.
  require
    is_ready: is_ready
deferred
  ensure
    last_product_not_void: last_product /= Void
end
class interface  
  TWO_PART_BUILDER [F -> BUILDABLE, G, H] 
inherit  
  BUILDER [F] 
create  
  make 
feature {NONE} -- Initialization 
  make (f: like factory_function_f; g: like factory_function_g;  
   h: like factory_function_h) 
    -- Set factory_function_f to f. Set factory_function_g to g.  
    -- Set factory_function_h to h. 
require  
  f_not_void: f /= Void  
  g_not_void: g /= Void  
  h_not_void: h /= Void 
ensure  
  factory_function_f_set: factory_function_f = f  
  factory_function_g_set: factory_function_g = g  
  factory_function_h_set: factory_function_h = h 
feature -- Access  
  last_product: F 
    -- Product under construction
Two-part builder (2/4)

**feature** -- Status report

*is_ready*: BOOLEAN

-- Is builder ready to build *last_product*?

**valid_args** (*args_f, args_g, args_h*: TUPLE): BOOLEAN

-- Are *args_f*, *args_g* and *args_h* valid arguments to

**feature** -- Basic operations

*build is*

-- Build *last_product*. (Successively call *build_g* and

**do**

*last_product* := *f_factory*.new

*build_g* ([[]])

*build_h* ([[]])

**ensure then**

*g_not_void*: *last_product*.g /= Void

*h_not_void*: *last_product*.h /= Void

**end**
Two-part builder (3/4)

\[\text{build}_\text{with}_\text{args} \ (\text{args}_f, \text{args}_g, \text{args}_h: \ TUPLE)\]
\[\text{-- Build } \text{last}_\text{product} \text{ with } \text{args}_f. \text{ (Successively} \]
\[\text{-- call } \text{build}_g \text{ with } \text{args}_g \text{ and } \text{build}_h \text{ with} \]
\[\text{-- args}_h \text{ to build product parts.})\]

**require**
\[\text{valid}_\text{args}: \text{valid}_\text{args} \ (\text{args}_f, \text{args}_g, \text{args}_h)\]

**ensure**
\[\text{g}_\text{not}_\text{void}: \text{last}_\text{product}.g /= \text{Void}\]
\[\text{h}_\text{not}_\text{void}: \text{last}_\text{product}.h /= \text{Void}\]

**feature** -- Factory functions

\[\text{factory}_\text{function}_f: \text{FUNCTION} \ [\text{ANY, TUPLE, } F]\]
\[\text{-- Factory function creating new instances of type } F\]

\[\text{factory}_\text{function}_g: \text{FUNCTION} \ [\text{ANY, TUPLE, } G]\]
\[\text{-- Factory function creating new instances of type } G\]

\[\text{factory}_\text{function}_h: \text{FUNCTION} \ [\text{ANY, TUPLE, } H]\]
\[\text{-- Factory function creating new instances of type } H\]
Two-part builder (4/4)

**feature** \{NONE\} -- Basic operations

\( \text{build}_g \) (\( \text{args}_g \): TUPLE) \textbf{is} ...

\( \text{build}_h \) (\( \text{args}_h \): TUPLE) \textbf{is} ...

**feature** \{NONE\} -- Factories

\( \text{f}_\text{factory} \): FACTORY [\( F \)]

-- Factory of objects of type \( F \)

\( \text{g}_\text{factory} \): FACTORY [\( G \)]

-- Factory of objects of type \( G \)

\( \text{h}_\text{factory} \): FACTORY [\( H \)]

-- Factory of objects of type \( H \)

**invariant**

\( \text{factory}_\text{function}_f \) \_not\_void: \( \text{factory}_\text{function}_f \) \( = \) \textbf{Void}

\( \text{factory}_\text{function}_g \) \_not\_void: \( \text{factory}_\text{function}_g \) \( = \) \textbf{Void}

\( \text{factory}_\text{function}_h \) \_not\_void: \( \text{factory}_\text{function}_h \) \( = \) \textbf{Void}

\( \text{f}_\text{factory} \) \_not\_void: \( \text{f}_\text{factory} \) \( = \) \textbf{Void}

\( \text{g}_\text{factory} \) \_not\_void: \( \text{g}_\text{factory} \) \( = \) \textbf{Void}

\( \text{h}_\text{factory} \) \_not\_void: \( \text{h}_\text{factory} \) \( = \) \textbf{Void}

end
Example using a two-part builder

class APPLICATION
create
feature {NONE} -- Initialization
  make is
    -- Build a new two-part product with a two-part builder.
    local
      my_builder: TWO_PART_BUILDER [TWO_PART_PRODUCT,
        PART_A, PART_B]
      my_product: TWO_PART_PRODUCT
    do
      create my_builder.make (agent new_product, agent new_part_a, agent new_part_b)
      my_builder.build_with_args (["Two-part product"],["Part A"],["Part B"])
      my_product := my_builder.last_product
    end
feature -- Factory functions
  new_product (a_name: STRING): TWO_PART_PRODUCT is ...
  new_part_a (a_name: STRING): PART_A is ...
  new_part_b (a_name: STRING): PART_B is ...
end
Two-part builder

class

\texttt{TWO\_PART\_BUILDER} \[ F \rightarrow \texttt{BUILDABLE}, G, H \]

\begin{itemize}
  \item \( F \): type of product to build
  \item \( G \): type of first part of the product
  \item \( H \): type of second part of the product
\end{itemize}

\( \Rightarrow \) The builder knows the type of product to build and number of parts

\begin{itemize}
  \item In the original Builder pattern:
    \begin{itemize}
      \item Deferred builder does not know the type of product to build
      \item Concrete builders know the type of product to build
    \end{itemize}
  \item \texttt{TWO\_PART\_BUILDER} is a concrete builder
    \( \Rightarrow \) compatible with the pattern
class
TWO_PART_BUILDER [F -> BUILDABLE, G, H]
inherit
BUILDER [F]

... feature -- Factory functions
factory_function_f: FUNCTION [ANY, TUPLE, F]
    -- Factory function creating new instances of type F
factory_function_g: FUNCTION [ANY, TUPLE, G]
    -- Factory function creating new instances of type G
factory_function_h: FUNCTION [ANY, TUPLE, H]
    -- Factory function creating new instances of type H

feature {NONE} -- Implementation
build_g (args_g: TUPLE) is
    -- Set last_product.g with a new instance of type G created with
    -- arguments args_g.
    do
        last_product.set_g (g_factory.new_with_args (args_g))
    end
end

Chair of Software Engineering
Builder Library: completeness?

- Supports builders that need to create two-part or three-part products

- Cannot know the number of parts of product to be built in general

⇒ Incomplete support of the Builder pattern ("Componentizable but non-comprehensive")
Builder: Componentization outcome

- **Completeness**
  - Some typical cases of the Builder pattern: two-part and three-part products (“Componentizable but not comprehensive”)

- **Usefulness**
  - Reusable
  - Easy-to-use

- **Faithfulness**
  - Similar to an implementation of Builder (with genericity and agents)

- **Type-safety**
  - Type-safe (constrained genericity, agents, Factory Library)

- **Performance**
  - Same order as the Builder pattern

- **Extended applicability**
  - No more cases
Agenda for today

- Builder
- Proxy
- State
  - Original pattern and variants
  - Componentization
“Describe[s] how to provide a surrogate or placeholder for another object to control access to it.” [GoF, p 207]
Flaws of the approach

- No reusable solution
- Only one kind of proxies: “virtual proxy”
Componentizability classification

1. Componentizable
   1.1 Built-in
   1.2 Library-supported
   1.3 Newly componentized
   1.4 Possible component

   1.3.1 Fully componentizable
   1.3.2 Componentizable but not comprehensive
   1.3.3 Componentizable but unfaithful
   1.3.4 Componentizable but useless

   Prototype
   Flyweight
   Observer
   Mediator
   Abstract Factory
   Factory Method
   Visitor
   Command
   Composite
   Chain of Responsibility

   Builder
   Proxy
   State

Design pattern

2. Non-componentizable
   2.1 Skeleton
   2.2 Possible skeleton
   2.3 Some library support
   2.4 Design idea

   Singleton
   Iterator
   Facade
   Interpreter

   Decorator
   Adapter
   Template Method
   Bridge

   2.1.1 Method
   2.1.2 No method

   Chain of Responsibility
   Visitor
   Command
   Composite
   Bridge
Proxy Library

APPLICATION

* SUBJECT

characteristic*
set_characteristic*
request*
request_with_args*

actual_subject

REAL_SUBJECT

PROXY

[G -> SUBJECT create make end]

characteristic+
set_characteristic+
request+
request_with_args+

cached_characteristic
Mechanisms enabling componentization: constrained genericity

```plaintext
class interface
    PROXY \[G \rightarrow SUBJECT\) create make end]

inherit
    SUBJECT

create
    make

feature \{NONE\} -- Initialization
    make (a_characteristic: like characteristic)
    -- Initialize subject with a_characteristic.
    ensure then
        cached_characteristic_set: cached_characteristic = a_characteristic

feature -- Access
    characteristic: TUPLE
    -- Characteristic of a subject
    ensure then
        is_cached_characteristic: Result = cached_characteristic
```
Class PROXY (2/3)

feature -- Status report
valid_args (args: TUPLE): BOOLEAN
    -- Are args valid arguments for request_with_args?
ensure
definition: Result = subject.valid_args (args)

feature -- Basic operations
request is
    -- Request something on current subject.
    do
        subject.request
    end

request_with_args (args: TUPLE)
    -- Request something on current subject using args.
require
    valid_args: valid_args (args)

feature -- Status setting
set_characteristic (a_characteristic: like characteristic)
    -- Set characteristic to a_characteristic.
ensure then
    cached_characteristic_set: cached_characteristic = a_characteristic
Class PROXY (3/3)

feature {NONE} -- Implementation
   actual_subject: G
      -- Actual subject (loaded only when needed)
   subject: G is
      -- Subject
      do
         if actual_subject = Void then
            create actual_subject.make (cached_characteristic)
            cached_characteristic := actual_subject.characteristic
         end
      Result := actual_subject
   ensure
      subject_not_void: Result /= Void
      is_actual_subject: Result = actual_subject
      cached_characteristic_not_void: cached_characteristic /= Void
   end
   cached_characteristic: like characteristic
      -- Cache of characteristic of actual subject
   invariant
      cached_characteristic_not_void: cached_characteristic /= Void
      consistent: actual_subject /= Void implies
         cached_characteristic = actual_subject.characteristic
   end
deferred class SUBJECT
feature {NONE} -- Initialization
  make (a_characteristic: like characteristic) is
    -- Initialize subject with a_characteristic.
    require
    a_characteristic_not_void: a_characteristic /= Void
  deferred end
feature -- Access
  characteristic: TUPLE is
    -- Characteristic of a subject
    deferred ensure
    characteristic_not_void: Result /= Void
  end
feature -- Status report
  valid_args (args: TUPLE): BOOLEAN is
    -- Are args valid arguments for request_with_args?
  deferred end
**Class SUBJECT (2/2)**

```plaintext
feature -- Status setting
    set_characteristic (a_characteristic: like characteristic) is
        -- Set characteristic to a_characteristic.
        require
            a_characteristic_not_void: a_characteristic /= Void
        deferred
        ensure
            characteristic_set: characteristic = a_characteristic
    end

feature -- Basic operations
    request is
        -- Request something on current subject.
        require
            characteristic_not_void: characteristic /= Void
        deferred
    end

    request_with_args (args: TUPLE) is
        -- Request something on current subject using args.
        require
            characteristic_not_void: characteristic /= Void
            valid_args: valid_args (args)
        deferred
    end
```

*Chair of Software Engineering*
Proxy Library: **SUBJECT** is fully-deferred!??

- **APPLICATION** → **SUBJECT**
  - characteristic*
  - set_characteristic*
  - request*
  - request_with_args*

- **REAL_SUBJECT**
  - characteristic*
  - set_characteristic*
  - request*
  - request_with_args*

- **PROXY**
  - actual_subject
  - characteristic*
  - set_characteristic*
  - request*
  - request_with_args*
  - cached_characteristic

---

**Library**

```cpp
class PROXY [G -> SUBJECT create make end]
```

- In both cases: Clients need to implement the class corresponding to the object to be turned into a proxy.

- What the library brings:
  - It encapsulates the **proxy machinery** into a reusable class **PROXY**
  - It provides an interface (≈ a mould) to use the Proxy Library
Different kinds of proxies (1/2)

- Smart references
  - Requires the ability to redefine the dot operator (e.g. to add reference counting)
  - Could be simulated with the Proxy Library

- Remote proxies
  - Subject and proxy may be on different physical machines
  - Requires knowing the inter-process communication mechanism (e.g. CORBA)
Different kinds of proxies (2/2)

- Protection proxies
  - Give objects different access rights
  - Could extend `request` and `request_with_args`:

```plaintext
class PROXY [G -> SUBJECT create make end]
inherit SUBJECT
...
feature -- Basic operations
  request is
    do
      if some_access_rights then
        subject.request
      elseif some_other_access_rights then
        ...
    end
  end

-- Request something on current subject.

Requires context information

Not componentizable
```
Proxy: Componentization outcome

- Completeness
  - Some cases of the Proxy pattern: “virtual proxies”
    ⇒ “Componentizable but not comprehensive”

- Usefulness
  - Reusable
  - Easy-to-use

- Faithfulness
  - Similar to a traditional implementation of Proxy (with genericity)

- Type-safety
  - Type-safe (constrained genericity)

- Performance
  - Same order as the Proxy pattern

- Extended applicability
  - No more cases
Agenda for today

- Builder
- Proxy
- State
  - Original pattern and variants
  - Componentization
### Original State pattern

**Without the State pattern**

```plaintext
class BOOK
...
feature -- Basic operation
  borrow is
    -- Borrow book.
    do
      if free then
        borrowed := True
        free := False
      elseif borrowed then
        -- Display a message
        -- to the user.
      end
    end
end
```

**With the State pattern**

```plaintext
class BOOK
...
feature -- Basic operation
  borrow is
    -- Borrow book.
    do
      state.borrow
    end
elseif borrowed then
  -- Display a message
  -- to the user.
end
end
```
Seven State variants [Dyson 1996]

Cases covered by each pattern

- State
  - State attribute
  - Pure State
  - Exported State
  - State-driven transitions
  - Context-driven transitions
  - Default State
State attribute

- Explains where to put attributes:
  - If an attribute only makes sense in one particular state, put it in the corresponding state class.
    - e.g. *user* in class *BORROWED*
  - If an attribute makes sense in several – but not all – states, put it in a common ancestor of the corresponding state classes.
  - If an attribute is state-independent, put it in the context class.
    - e.g. *reservations*: *LIST [RESERVATION]*
Pure State

- Applicable when the \textit{STATE} classes have no attribute.

$\Rightarrow$ \textit{STATE} objects can be \textit{shared} between different contexts.
Exported State

- Export the attribute *state* of the context class to any client to avoid multiplying proxy routines.
- Interesting when the *STATE* classes have many attributes.

```plaintext
class BOOK

... feature -- Access (exported to ANY)
    state: STATE
        -- Current book state

... end

class BOOK_CLIENT

... feature -- Access

    book: BOOK
        -- Book
    user: PERSON is
        -- Current user of book
    do
        Result := book.state.user
    end

... end
```
State-driven transitions

- State objects are responsible for changing the context’s state. (The \textit{STATE} classes implement the automaton.)

```plaintext
class 
  FREE
inherit 
  STATE
...
feature -- Basic operation
  borrow is
    -- Borrow book. (Create a new state \textit{BORROWED}
    -- and set it to the book.)
    do
      book.set_state (create \{\textit{BORROWED}\}.make (book))
    end
end
```
Context-driven transition

- Context is responsible for changing state.
- Useful to reuse the same state objects with different contexts.

```ruby
class VIDEO_RECORDER
  ...
  feature -- Basic operation
    return is
      -- Return video recorder.
      -- Create a new state MAINTAINED and set it.
      do
        set_state (create {MAINTAINED}.make (Current))
      end
  ...
end
```
Function `default_state` in the context class called by the creation procedure to ensure the created context is consistent.

```literate
class BORROWABLE
create
make
feature {NONE} -- Initialization
make is
    -- Set initial state.
do
    set_state (default_state)
ensure
    default_state_set:
        state = default_state
end

feature {NONE} -- Implementation
    default_state: STATE is
        -- Default state
deferred
ensure
default_state_not_void:
    Result /= Void
end
... end
```
Agenda for today

- Builder
- Proxy
- State
  - Original pattern and variants
  - Componentization
“Allow[s] an object to alter its behavior when its internal state changes. The object will appear to change its class.” [GoF, p 305]
Delegating work to the state object

class
    CONTEXT
...

feature -- Basic operation
    do_something is
        -- Do something depending on the state.
        do
            state.do_something
        end
end

feature {NONE} -- Implementation
    state: STATE
        -- Current state
...
end
class INITIAL_STATE
inherit STATE

... feature -- Basic operation
  do_something is
    -- Do something depending on the state.
    do
      -- Do something.
      context.set_state (context.intermediary_state)
    end
  end

... end
Componentizability classification

1. Componentizable
   1.1 Built-in
   1.2 Library-supported
   1.3 Newly componentized
   1.4 Possible component
      1.3.1 Fully componentizable but not comprehensive
         Flyweight
         Observer
         Mediator
         Abstract Factory
         Factory Method
         Visitor
         Command
         Composite
         Chain of Responsibility
      1.3.2 Componentizable but not comprehensive
         Builder
         Proxy
         State
      1.3.3 Componentizable but unfaithful
         Strategy
      1.3.4 Componentizable but useless
         Memento

2. Non-componentizable
   2.1 Skeleton
   2.2 Possible skeleton
   2.3 Some library support
   2.4 Design idea
      2.1.1 Method
      2.1.2 No method
      2.1.3 Some library support
      2.1.4 Design idea
         Singleton
         Iterator
         Facade
         Interpreter
      2.2.1 Prototype
      2.2.2 Flyweight
      2.2.3 Mediator
      2.2.4 Abstract Factory
      2.2.5 Factory Method
      2.2.6 Visitor
      2.2.7 Command
      2.2.8 Composite
      2.2.9 Chain of Responsibility
      2.3.1 Prototype
      2.3.2 Flyweight
      2.3.3 Mediator
      2.3.4 Abstract Factory
      2.3.5 Factory Method
      2.3.6 Visitor
      2.3.7 Command
      2.3.8 Composite
      2.3.9 Chain of Responsibility
      2.4.1 Prototype
      2.4.2 Flyweight
      2.4.3 Mediator
      2.4.4 Abstract Factory
      2.4.5 Factory Method
      2.4.6 Visitor
      2.4.7 Command
      2.4.8 Composite
      2.4.9 Chain of Responsibility

Design pattern

Chair of Software Engineering

Trusted Components: Reuse, Contracts and Patterns - Lecture 18
Mechanisms enabling componentization:
simple inheritance, contracts

CONTEXT

STATE

STATE

STATE

INITIAL_STATE

INTERMEDIARY_STATE

FINAL_STATE

CONTEXT

STATE

STATE

STATE

STATE

State Library
Class CONTEXT (1/2)

class CONTEXT
create
make

feature {NONE} -- Initialization
make is
  -- Initialize state to a "null" state that does nothing.
do
create {NULL_STATE} state.make (Current)
ensure
  null_state: state.conforms_to (create {NULL_STATE}.make (Current))
end

feature -- Basic operations
  do something is
    -- Do something depending on the state.
do
    state.do_something
end
Class CONTEXT (2/2)

**feature -- Access**

**state: STATE**

-- Current state of the application

**feature -- Element change**

**set_state (a_state: like state) is**

-- Set state to a_state.

**require**

a_state_not_void: a_state /= Void

**do**

state := a_state

**ensure**

state_set: state = a_state

**end**

**invariant**

state_not_void: state /= Void

**end**
Why initializing the state context with a `NULL_STATE` instead of `Void`?

```plaintext
class CONTEXT
create
make

... feature -- Basic operations
  do_something is
    do
      state.do_something
    end
  end

feature -- Access
  state: STATE
    -- Current state of the application

... invariant
  state_not_void: state /= Void
  end
```

The creation procedure must produce an object that satisfies the invariant.

The invariant is needed because we do not want to have preconditions everywhere saying the `state` should not be `Void`. 
deferred class

STATE

feature {NONE} -- Initialization

make (a_context: like context) is

  -- Set context to a_context.

  require
  a_context_not_void: a_context /= Void

do
  context := a_context

ensure
  context_set: context = a_context
end

make_with_next (a_context: like context; a_state: like next) is

  -- Set context to a_context and next to a_state.

  require
  a_context_not_void: a_context /= Void
  a_state_not_void: a_state /= Void

do
  context := a_context
  next := a_state

ensure
  context_set: context = a_context
  next_set: next = a_state
end
feature -- Access
context: CONTEXT

   -- Application context
next: STATE

   -- Next state

feature -- Status setting
set_next (a_state: like next) is

   -- Set next to a_state.
do
   next := a_state
ensure
   next_set: next = a_state
end

feature -- Basic operations
do_something is

   -- Do something depending on the state.
do
   do_something_imp
   if next /= Void then
      context.set_state (next)
   end
ensure
   next_state_set: next /= Void implies context.state = next
end
Class *STATE* (3/3)

```plaintext
feature {NONE} -- Implementation

  do_something_imp is
    -- Do something depending on the state.
    deferred
    end

invariant
  context_not_void: context /= Void
end
```
Is the State Library flexible enough to support cases when the next state depends on arguments received by feature `do_something`?
deferred class  
STATE

...  

feature -- Basic operations
  do_something is
    -- Do something depending on the state.
    do
      do_something_imp
      if next /= Void then
        context.set_state (next)
      end
    end
  ensure
    next_state_set: next /= Void implies context.state = next
  end

feature {NONE} -- Implementation
  do_something_imp is
    -- Do something depending on the state.
    deferred
  end

invariant
  context_not_void: context /= Void
end
State: Componentization outcome

- Completeness
  - Some cases of the State pattern: “state-driven transitions”
  - “Componentizable but not comprehensive”

- Usefulness
  - Reusable
  - Easy-to-use

- Faithfulness
  - Similar to a traditional implementation of the State pattern (with inheritance, assertions)

- Type-safety
  - Type-safe (inheritance, assertions)

- Performance
  - Same order as the State pattern

- Extended applicability
  - No more cases
Language support

- **Delegation-based languages** (e.g. Self) support the State pattern natively (they enable changing an object’s state at run time)

- However, the Self approach has some drawbacks:
  - Modifying a program “on the fly” may be misused; one should rather encourage programmers to think about core abstractions
  - Self is dynamically typed; no error detected at compile time, which is dangerous
Complementary material (1/2)

- **From Patterns to Components:**
  - Chapter 13: Builder, Proxy and State

- **Further reading:**
  - Erich Gamma et al.: *Design Patterns*, 1995. (Builder, p 97-106; Proxy, p 207-218; State, p 305-314)
Further reading:


End of lecture 18